We resume @ 1pm

ATPESC

(Argonne Training Program on Extreme-Scale Computing)

A Performance Tuning Methodology: From the System Down to the Hardware

James Reinders, Intel

August 3, 2015, Pheasant Run, St Charles, IL

13:00-13:45

ATPESC

(Argonne Training Program on Extreme-Scale Computing)

A Performance Tuning Methodology: From the System Down to the Hardware

James Reinders, Intel

August 3, 2015, Pheasant Run, St Charles, IL

13:00-13:45

It is hard to “see” if you do not look.

It is hard to "see" if you do not look.

We could guess,
after all – we are smart enough
to *believe* we know what is happening.


Look for:
?

Look for:

- Confirmation


Look for:
• Confirmation
• Surprises

Look for:

- Confirmation
- Surprises

Your EXPERTISE will grow as you investigate.

Tools and Concepts

Tools:

Intel® VTune™ Amplifier	Profiling, node level counter analysis
Intel® Trace Analyzer and Collector	MPI, cluster level communication analysis
Intel® Inspector	Threading, Memory issues (node level)
Intel® Advisor	Scaling and Vectorization analysis and advice (node level)

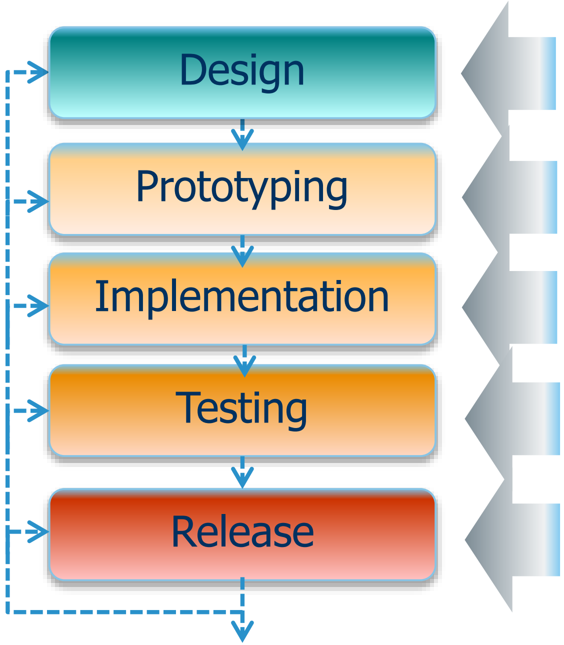
Why performance profiling?

Project performance tuning for:

- Reducing direct compute time costs
- Decreasing indirect costs
- Better user/customer experience

If you are not in that business, don't bother

Project development cycle and performance analysis



Think performance wise (app/sys level)

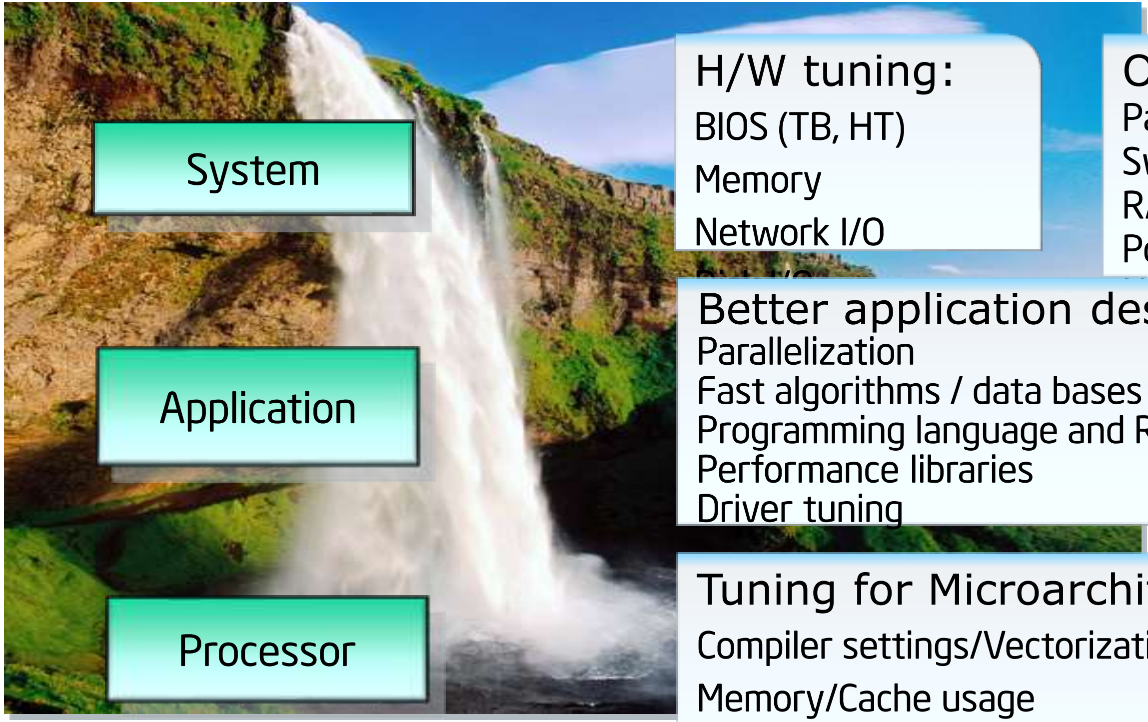
Choose perf. effective solutions

Apply perf. optimization and check results

Add perf. regression phase to test stage

Collect and analyze perf. related complaints from users of your product

Optimization: A Top-down Approach



H/W tuning:
BIOS (TB, HT)
Memory
Network I/O

OS tuning:
Page size
Swap file
RAM Disk
Power settings

Better application design:
Parallelization
Fast algorithms / data bases
Programming language and RT libs
Performance libraries
Driver tuning

Tuning for Microarchitecture:
Compiler settings/Vectorization
Memory/Cache usage
CPU pitfalls

OS, System
Expertise
SW/uArch

https://software.intel.com/en-us/articles/de-mystifying-software-performance-optimization

Performance profiling tools

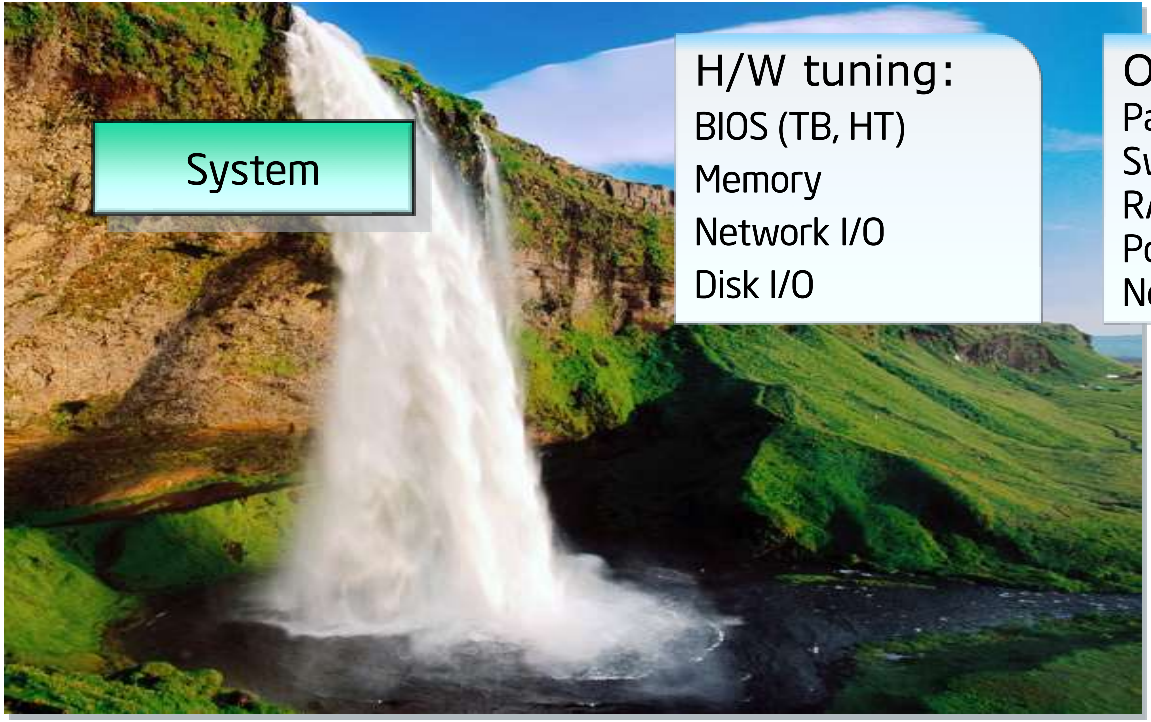
Level wise selection

System	<u>System profiler</u>	<u>OS embedded</u>
	Universal (for OS, HW)	Windows: Perf mon, Proc mon
	Proprietary (OS+HW)	Linux: top, vmstat, OProfile
Application	<u>Supported languages</u>	Windows: WPT, Xperf, VTune
IDE based	.Net/C#, Java	Managed: .Net, Java tools, VTune
Command Line	Python, Java Script, HTML	Linux: gprof, Valgrind, Google perftools, Crxprof, VTune
	C, C++, Fortran	

Microarchitcture

Provided by CPU/Platform manufacturer

Optimization: A Top-down Approach



H/W tuning:

BIOS (TB, HT)
Memory
Network I/O
Disk I/O

OS tuning:

Page size
Swap file
RAM Disk
Power settings
Network protocols

OS, System

System Tuning

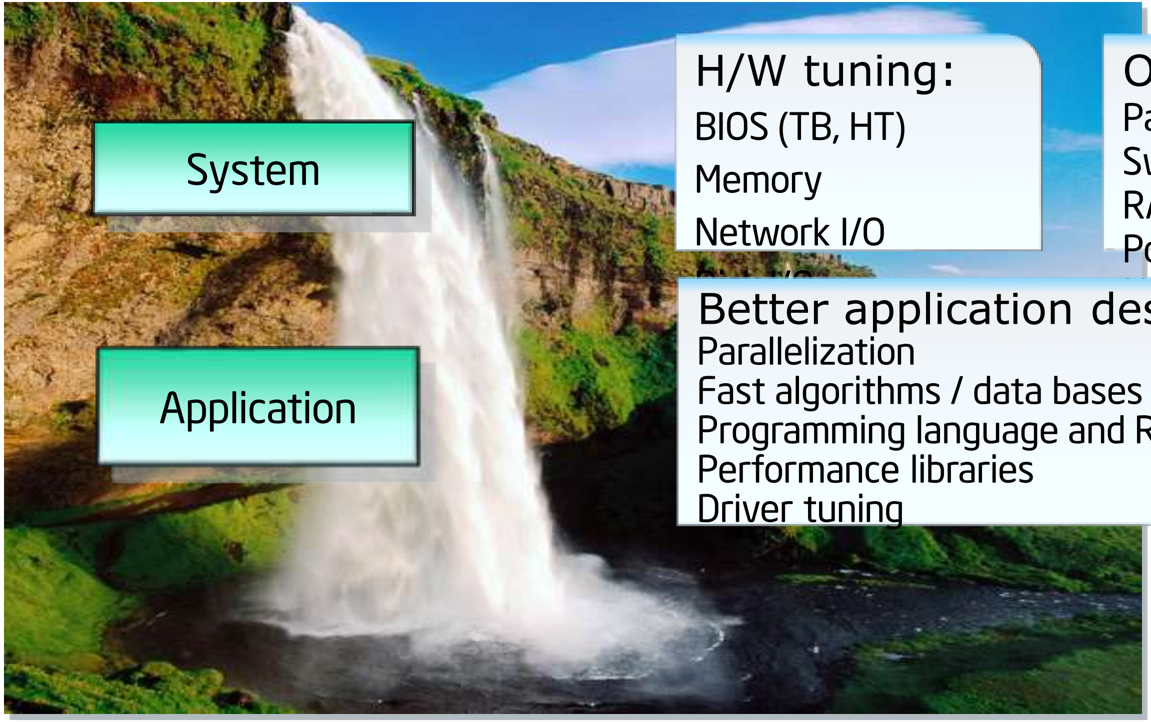
Who: System Administrators, Performance Engineers, Machine Owners, etc…

How:

- Benchmarks
  - Stream: www.cs.virginia.edu/stream/
  - Numerous FLOPS benchmarks
  - Network/MPI Benchmarks: www.intel.com/go/imb
  - <insert your favorite here>
- Tools
  - vmstat, top, sysprof, iostat, sar, Task Manager, etc…
  - Many vendor/platform specific tools

- Fixes
  - Upgrade Hardware - $$$
  - Check BIOS and OS configurations
    - Prefetchers, NUMA, Memory Configuration, Power Management, SMT

Optimization: A Top-down Approach



System

Application

H/W tuning:
BIOS (TB, HT)
Memory
Network I/O

OS tuning:
Page size
Swap file
RAM Disk
Power settings

Better application design:
Parallelization
Fast algorithms / data bases
Programming language and RT libs
Performance libraries
Driver tuning

OS, System   Expertise

Application Tuning

Who: Software Developers, Performance Engineers, Domain Experts

How:

- Workload selection
  - Repeatable results
  - Steady stat
- Define Metrics and Collect Baseline
  - Wall-clock time, FLOPS, FPS
  - <insert your metric here>
- Identify Hotspots
  - Focus effort where it counts
  - Use Tools
- Determine inefficiencies
  - Is there parallelism?
  - Are you memory bound?
  - Will better algorithms or programming languages help?

This step often requires some knowledge of the application and its algorithms

Application Tuning

Find Hotspots

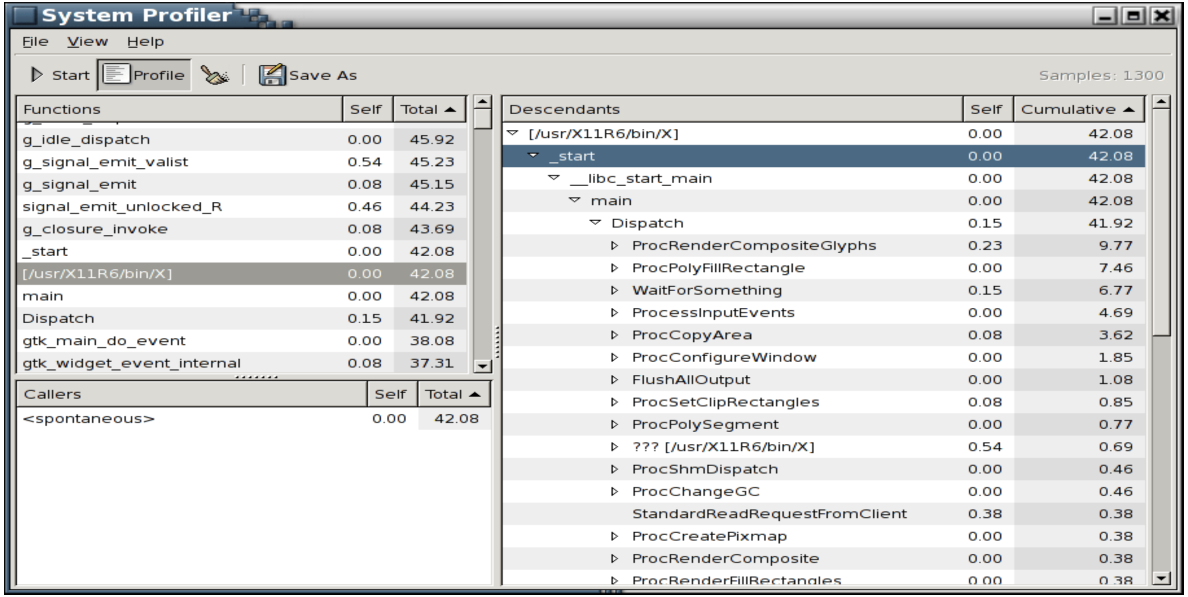
- This could be at the module, function, or source code level
- Determine your own granularity

```
$ opreport --exclude-dependent --demangle=smart --symbols `which lyx`
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask of 0x00 (No unit mask)
vma       samples  %          symbol name
081ec974 5016      8.5096     _Rb_tree<unsigned short, pair<unsigned short const, int>,  unsigned short
0810c4ec 3323      5.6375     Paragraph::getFontSettings(BufferParams const&, int) const
081319d8 3220      5.4627     LyXText::getFont(Buffer const*, Paragraph*, int) const
080e45d8 3011      5.1082     LyXFont::realize(LyXFont const&)
080e3d78 2623      4.4499     LyXFont::LyXFont()
081255a4 1823      3.0927     LyXText::singleWidth(BufferView*, Paragraph*, int, char) const
080e3cf0 1804      3.0605     operator==(LyXFont::FontBits const&, LyXFont::FontBits const&)
081128e0 1729      2.9332     Paragraph::Pimpl::getChar(int) const
081ed020 1380      2.3412     font_metrics::width(char const*, unsigned, LyXFont const&)
08110d60 1310      2.2224     Paragraph::getChar(int) const
081ebc94 1227      2.0816     qfont_loader::getfontinfo(LyXFont const&)
...
```

oprofile: http://oprofile.sourceforge.net/

Application Tuning
Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

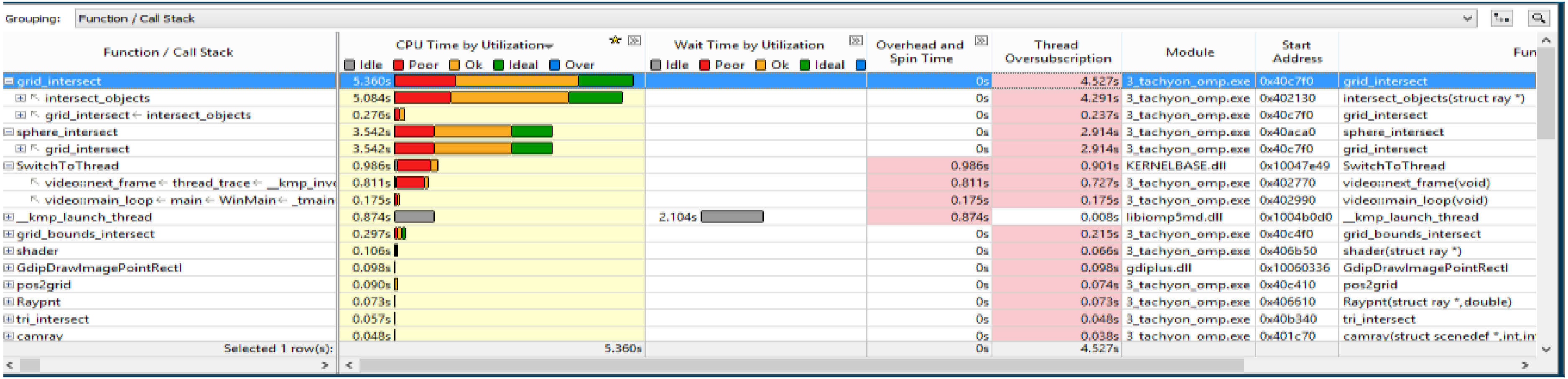


sysprof: http://sysprof.com

Application Tuning

Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity



Grouping: Function / Call Stack

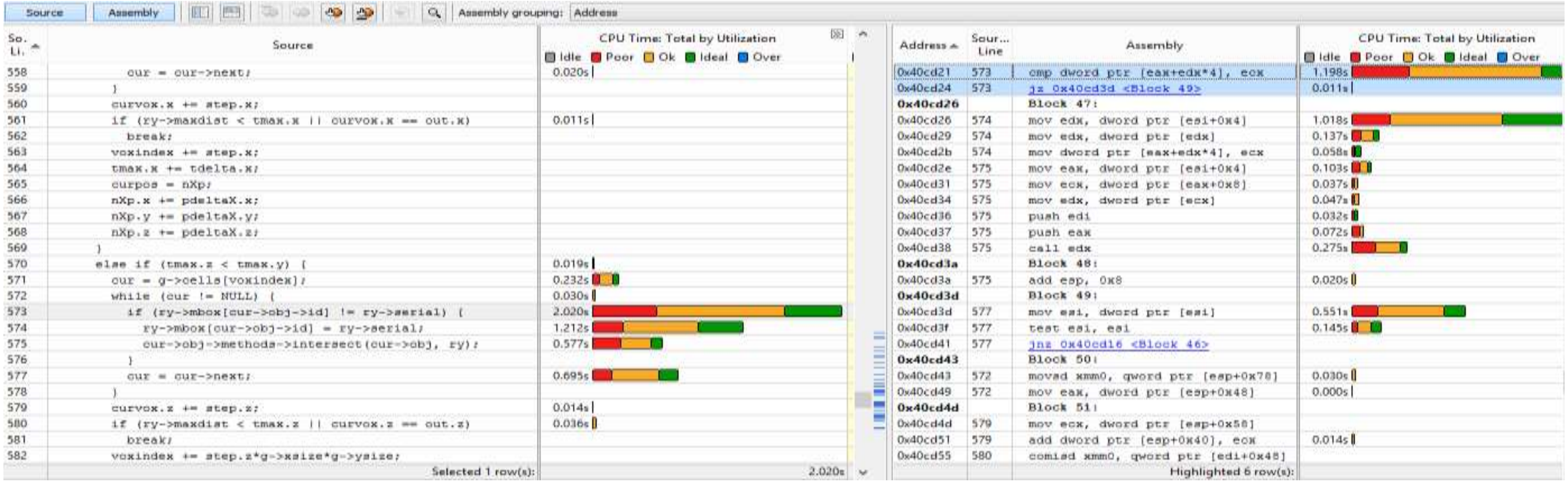
Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Time by Utilization (Idle, Poor, Ok, Ideal)	Overhead and Spin Time	Thread Oversubscription	Module	Start Address	Fun
grid_intersect	5.360s		0s	4.527s	3_tachyon_omp.exe	0x40c7f0	grid_intersect
intersect_objects	5.084s		0s	4.291s	3_tachyon_omp.exe	0x402130	intersect_objects(struct ray *)
grid_intersect ← intersect_objects	0.276s		0s	0.237s	3_tachyon_omp.exe	0x40c7f0	grid_intersect
sphere_intersect	3.542s		0s	2.914s	3_tachyon_omp.exe	0x40aca0	sphere_intersect
grid_intersect	3.542s		0s	2.914s	3_tachyon_omp.exe	0x40c7f0	grid_intersect
SwitchToThread	0.986s		0.986s	0.901s	KERNELBASE.dll	0x10047e49	SwitchToThread
video::next_frame ← thread_trace ← _kmp_invo	0.811s		0.811s	0.727s	3_tachyon_omp.exe	0x402770	video::next_frame(void)
video::main_loop ← main ← WinMain ← _tmain	0.175s		0.175s	0.175s	3_tachyon_omp.exe	0x402990	video::main_loop(void)
__kmp_launch_thread	0.874s	2.104s	0.874s	0.008s	libiomp5md.dll	0x1004b0d0	__kmp_launch_thread
grid_bounds_intersect	0.297s		0s	0.215s	3_tachyon_omp.exe	0x40c4f0	grid_bounds_intersect
shader	0.106s		0s	0.066s	3_tachyon_omp.exe	0x406b50	shader(struct ray *)
GdipDrawImagePointRectI	0.098s		0s	0.098s	gdiplus.dll	0x10060336	GdipDrawImagePointRectI
pos2grid	0.090s		0s	0.074s	3_tachyon_omp.exe	0x40c410	pos2grid
Raypnt	0.073s		0s	0.073s	3_tachyon_omp.exe	0x406610	Raypnt(struct ray *,double)
tri_intersect	0.057s		0s	0.048s	3_tachyon_omp.exe	0x40b340	tri_intersect
camray	0.048s		0s	0.038s	3_tachyon_omp.exe	0x401c70	camray(struct scenedef *,int,in
Selected 1 row(s):	5.360s		0s	4.527s			

Intel® VTune™ Amplifier XE: http://intel.ly/vtune-amplifier-xe

Application Tuning

Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

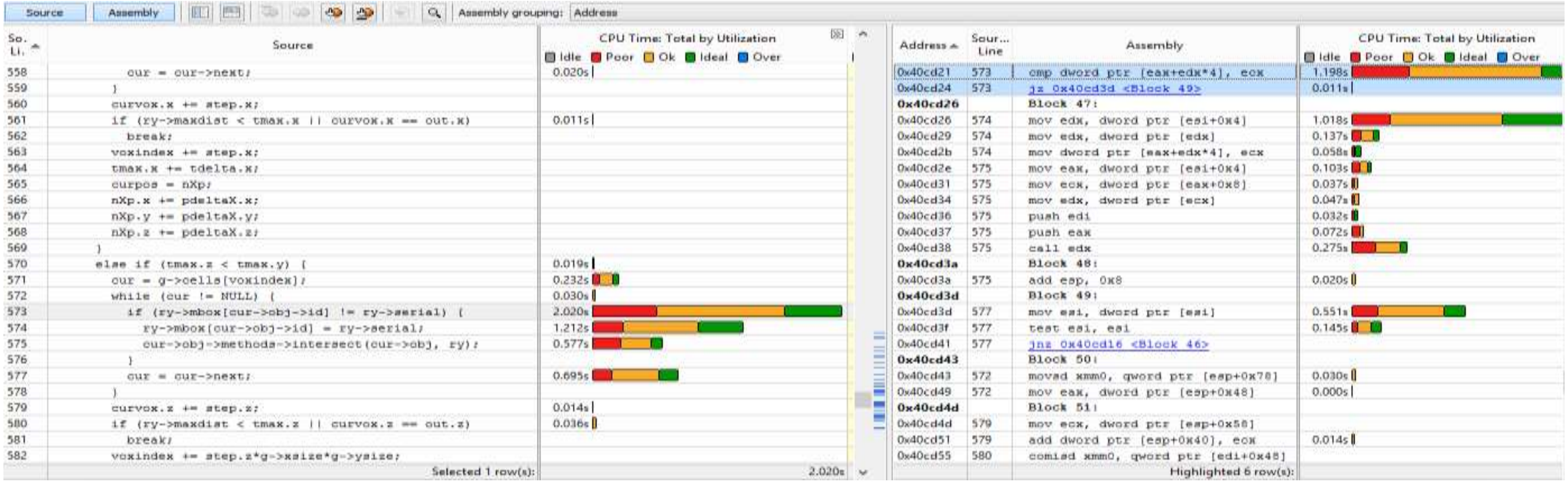


Intel® VTune™ Amplifier XE: http://intel.ly/vtune-amplifier-xe

Application Tuning

Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity



This may reinforce your understanding of the application but often reveals surprises

Application Tuning

Resource Utilization

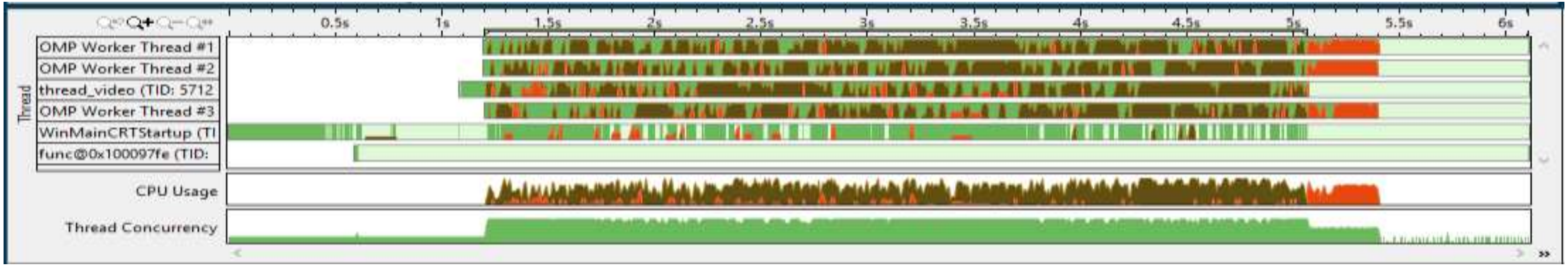
- Is the application parallel?
- Multi-thread vs. Multi-process
- Memory Bound?

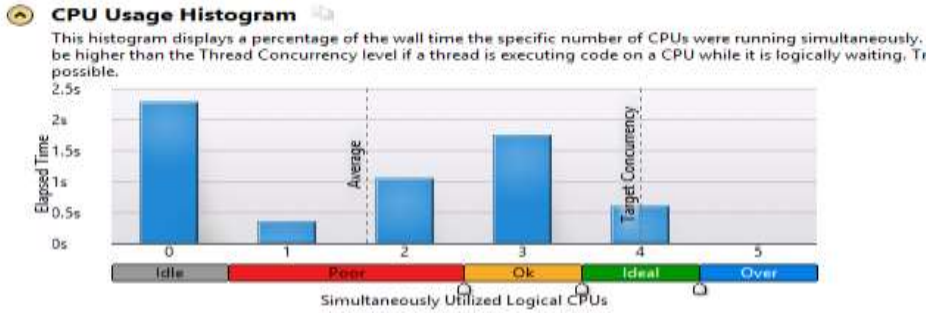
```
last pid: 86494;  load averages:  0.83,  0.65,  0.69   up 67+22:48:43  14:44:15
227 processes: 1 running, 224 sleeping, 2 zombie
CPU: 20.2% user,  0.0% nice,  6.5% system,  0.2% interrupt, 73.1% idle
Mem: 1657M Active, 1868M Inact, 273M Wired, 190M Cache, 112M Buf, 11M Free
Swap: 4500M Total, 249M Used, 4251M Free, 5% Inuse

  PID USERNAME   THR PRI NICE   SIZE    RES STATE  C   TIME   WCPU COMMAND
86460 www          1   4    0   150M 30204K accept 1   0:02 11.18% php-cgi
86458 www          1   4    0   150M 29912K accept 0   0:02  8.98% php-cgi
86463 pgsql        1   4    0   949M    99M sbwait 1   0:01  7.96% postgres
85885 www          1   4    0   150M 35204K accept 2   0:07  7.57% php-cgi
85274 www          1   4    0   149M 40868K sbwait 3   0:27  5.18% php-cgi
85267 www          1   4    0   151M 40044K sbwait 2   0:33  4.59% php-cgi
85884 www          1   4    0   150M 41584K accept 2   0:14  4.59% php-cgi
85887 pgsql        1   4    0   951M   128M sbwait 1   0:04  4.20% postgres
85886 pgsql        1   4    0   949M   161M sbwait 0   0:08  3.37% postgres
86459 pgsql        1   4    0   949M 75960K sbwait 2   0:01  3.37% postgres
85279 pgsql        1   4    0   950M   192M sbwait 2   0:14  2.39% postgres
85269 pgsql        1   4    0   950M   199M sbwait 1   0:19  2.20% postgres
85268 www          1   4    0   152M 44356K sbwait 2   0:32  1.17% php-cgi
85273 pgsql        1   4    0   950M   215M sbwait 0   0:19  1.17% postgres
97082 pgsql        1  44    0 26020K  6832K select 0  46:55  0.00% postgres
  892 root         1   4    0  3160K     8K -      2  13:33  0.00% nfsd
 1796 root         1  44    0 19780K 13660K select 3  12:43  0.00% Xvfb
```

Application Tuning

Resource Utilization

- Is the application parallel?





Elapsed Time: 6.107s

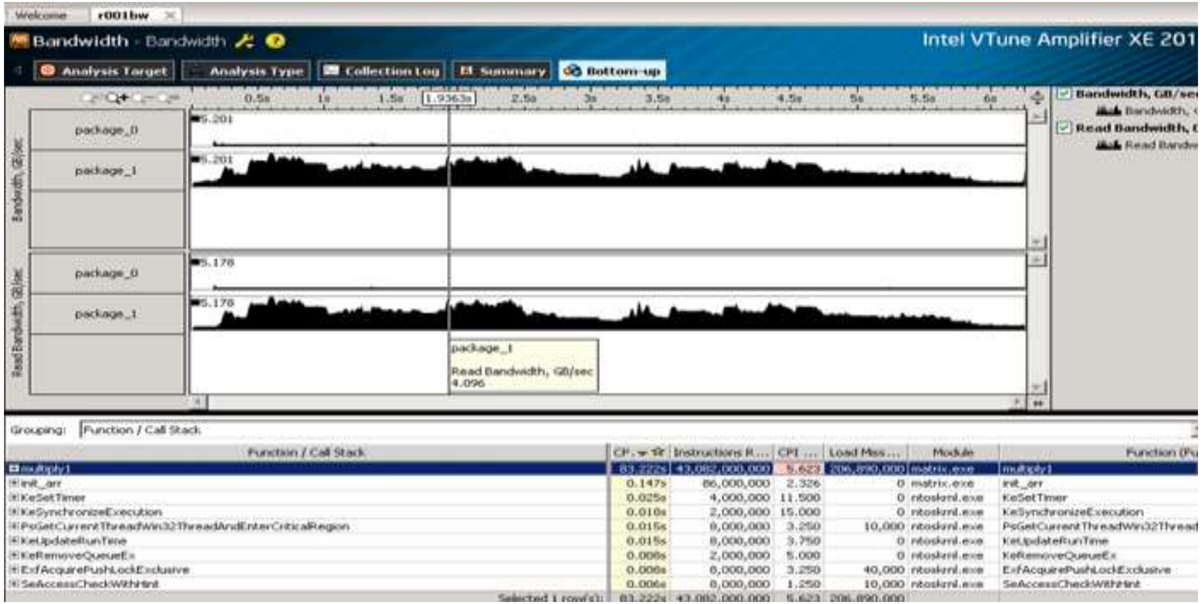
Total Thread Count:	6
Overhead Time:	0s
Spin Time:	1.909s

A significant portion of CPU time is spent waiting.
implementation (for example, by backing off then

CPU Time:	12.029s
Paused Time:	0s

Application Tuning
Resource Utilization

- Memory Bound?

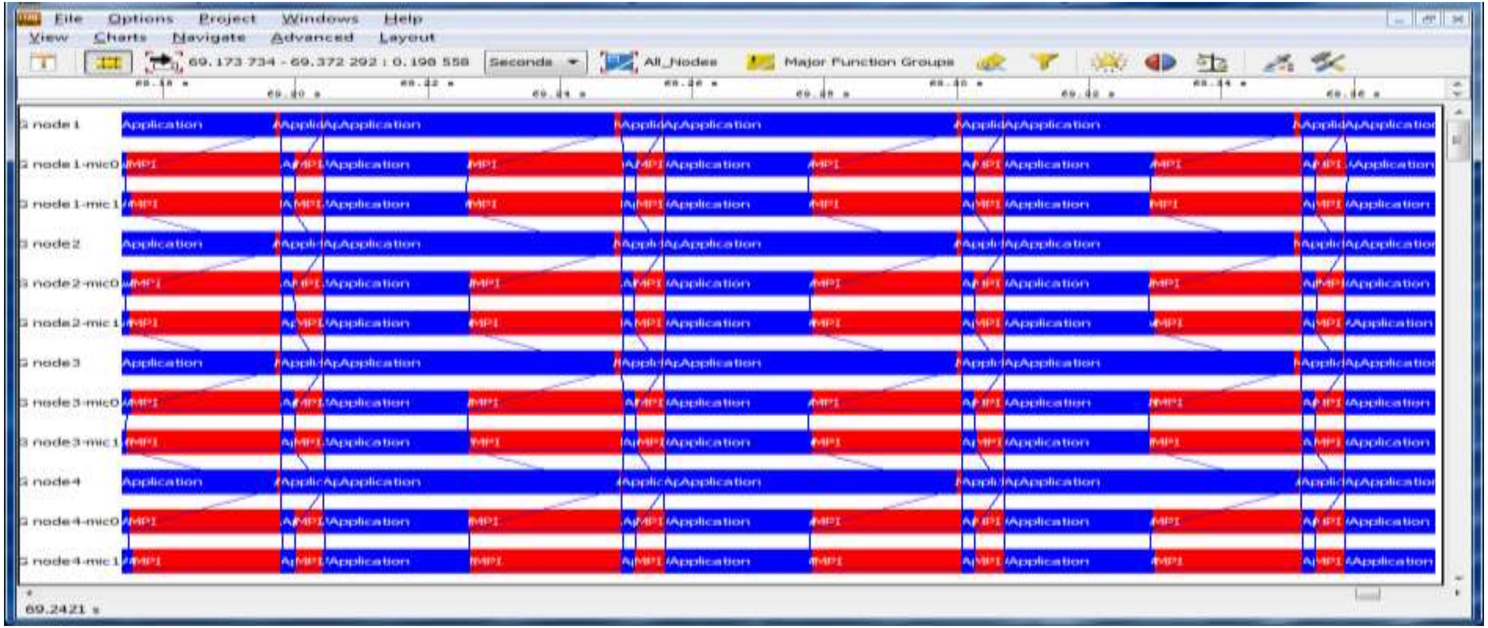


- Know your max theoretical memory bandwidth

Application Tuning

Resource Utilization

MPI applications have added communication complexity



Intel® Trace Analyzer and Collector: http://intel.ly/traceanalyzer-collector

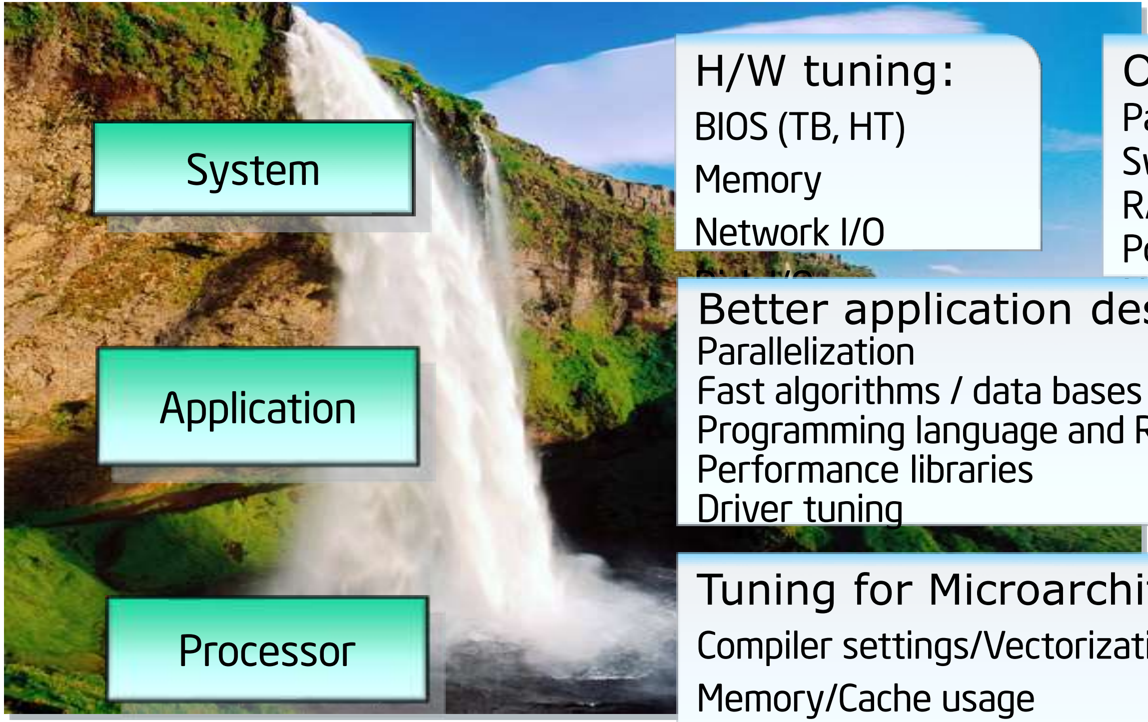
Application Tuning

What's Next?

- If your Hotspots are common algorithms:
  - Look for optimized libraries
- If your Hotspots are uncommon:
  - Compiler optimizations
  - Expert analysis and refactoring of an algorithm
    - The opposite of "low-hanging fruit"
  - Deeper analysis of hardware performance
    - More on this later
- If the system is underutilized:
  - Add parallelism - multi-thread or multi-process
    - OpenMP, TBB, Cilk, MPI, etc…

> Tools can help you determine where to look and may identify some issues.
> Some tools may provide suggestions for fixes.
> In the end – the developer and/or expert has to make the changes and decisions – there is no silver bullet.

Optimization: A Top-down Approach



H/W tuning:
BIOS (TB, HT)
Memory
Network I/O

OS tuning:
Page size
Swap file
RAM Disk
Power settings

Better application design:
Parallelization
Fast algorithms / data bases
Programming language and RT libs
Performance libraries
Driver tuning

Tuning for Microarchitecture:
Compiler settings/Vectorization
Memory/Cache usage
CPU pitfalls

OS, System
Expertise
SW/uArch

Microarchitecture Tuning

Who: ~~Architecture Experts~~
Software Developers, Performance Engineers, Domain Experts

How:

- Use architecture specific hardware events
- Use predefined metrics and best known methods
    - Often hardware specific
    - (Hopefully) provided by the vendor
- Tools make this possible for the non-expert
    - Linux perf
    - Intel® VTune™ Amplifier XE
- Follow the Top-Down Characterization
    - Locate the hardware bottlenecks
    - Whitepaper here: https://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues

Introduction to Performance Monitoring Unit (PMU)

Registers on Intel CPUs to count architectural events

- E.g. Instructions, Cache Misses, Branch Mispredict

Events can be counted or sampled

- Sampled events include Instruction Pointer

Raw event counts are difficult to interpret

- Use a tool like VTune or Perf with predefined metrics

Raw PMU Event Counts vs Metrics

Grouping: Function / Call Stack

Function / Call Stack	CPU_CL...	CPU_CLK_U...	INST_RETIRE...	L1D_PEND_...	OFF..	BR_MISP...	CPU_CLK_U...	CYCLE_AC...	CYCLE_AC...	DTL...	DTLB_LO...	DTLB_L...	DTL...	DTLB_ST...	DTLB_S...	ICACH..
grid_intersect	13,604,020,406	14,118,021,177	12,572,018,858	6,344,009,516	0	52,001,170	14,924,022,386	5,408,008,112	4,264,006,396	0	234,000,351	26,000,039	0	7,800,234	0	
sphere_intersect	8,706,013,059	9,134,013,701	8,494,012,741	4,238,006,357	0	15,600,351	9,464,014,196	3,016,004,524	2,808,004,212	0	104,000,156	26,000,039	0	10,400,312	0	
grid_bounds_intersect	984,001,476	1,004,001,506	672,001,008	104,000,156	0	15,600,351	962,001,443	312,000,468	286,000,429	0	0	0	0	0	0	
__kmp_end_split_barrier	676,001,014	624,000,936	460,000,690	0	0	0	0	0	0	0	0	0	0	0	0	
__kmp_x86_pause	228,000,342	224,000,336	122,000,183	0	0	10,400,234	0	0	0	0	0	0	0	0	0	
shader	216,000,324	242,000,363	142,000,213	104,000,156	0	0	208,000,312	104,000,156	52,000,078	0	0	0	0	2,600,078	0	
Raypnt	206,000,309	210,000,315	208,000,312	0	0	0	234,000,351	52,000,078	78,000,117	0	0	0	0	0	0	2,600,03
pos2grid	204,000,306	248,000,372	180,000,270	26,000,039	0	0	390,000,585	26,000,039	52,000,078	0	0	0	0	0	0	
tri_intersect	168,000,252	208,000,312	180,000,270	0	0	0	104,000,156	78,000,117	52,000,078	0	52,000,078	0	0	0	0	
VScale	124,000,186	126,000,189	164,000,246	0	0	0	234,000,351	52,000,078	0	0	0	0	0	0	0	
__kmp_yield	96,000,144	98,000,147	200,000,300	0	0	0	0	0	0	0	0	0	0	0	0	
Selected 1 row(s):	13,604,020,406	14,118,021,177	12,572,018,858	6,344,009,516	0	52,001,170	14,924,022,386	5,408,008,112	4,264,006,396	0	234,000,351	26,000,039	0	7,800,234	0	

Grouping: Function / Call Stack

Function / Call Stack	Clocktic...	Instructions Retired	CPI Rate	MUX Reliability	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)		
								Front-end Bound	
					Retiring	Bad Speculation	Back-End Bound	Front-End Latency	Front-End Bandwidth
grid_intersect	14,118,021,177	12,572,018,858	1.123	0.946	0.246	0.033	0.647	0.063	0.012
sphere_intersect	9,134,013,701	8,494,012,741	1.075	0.965	0.250	0.065	0.619	0.057	0.009
grid_bounds_intersect	1,004,001,506	672,001,008	1.494	0.958	0.227	0.000	0.715	0.104	0.000
__kmp_end_split_barrier	624,000,936	460,000,690	1.357	0.000	0.000	0.000	0.792	0.167	0.042
pos2grid	248,000,372	180,000,270	1.378	0.636	0.367	0.000	0.633	0.000	0.131
shader	242,000,363	142,000,213	1.704	0.860	0.322	0.000	0.946	0.000	0.027
__kmp_x86_pause	224,000,336	122,000,183	1.836	0.000	0.000	0.000	0.971	0.000	0.029
Raypnt	210,000,315	208,000,312	1.010	0.897	0.093	0.279	0.567	0.000	0.062
Selected 1 row(s):	14,118,021,177	12,572,018,858	1.123	0.946	0.246	0.033	0.647	0.063	0.012

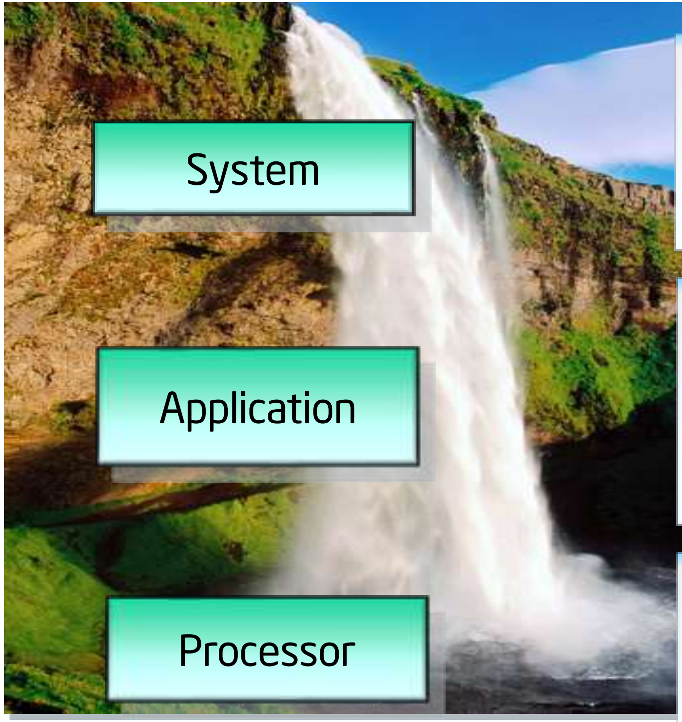
Adding Regression Tests for Performance

Regression testing isn't just for bugs

1. Create a baseline performance characterization
2. After each change or at a regular interval
    1. Compare new results to baseline
    2. Compare new results to previous results
    3. Evaluate the change
3. goto (1)

Performance tuning is easier if it's always on your mind and integrated into your development

Optimization: A Top-down Approach



H/W tuning:
- BIOS (TB, HT)
- Memory
- Network I/O

OS tuning:
- Page size
- Swap file
- RAM Disk
- Power settings

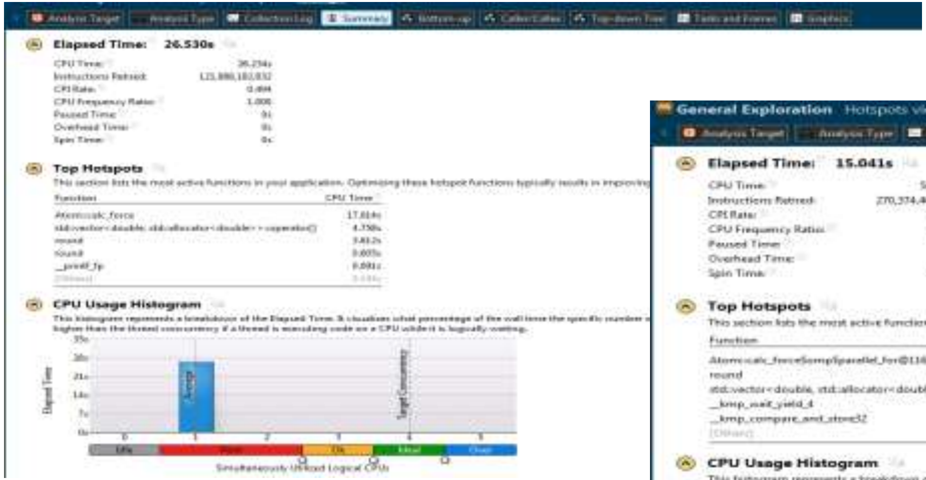
Better application design:
- Parallelization
- Fast algorithms / data bases
- Programming language and RT libs
- Performance libraries
- Driver tuning

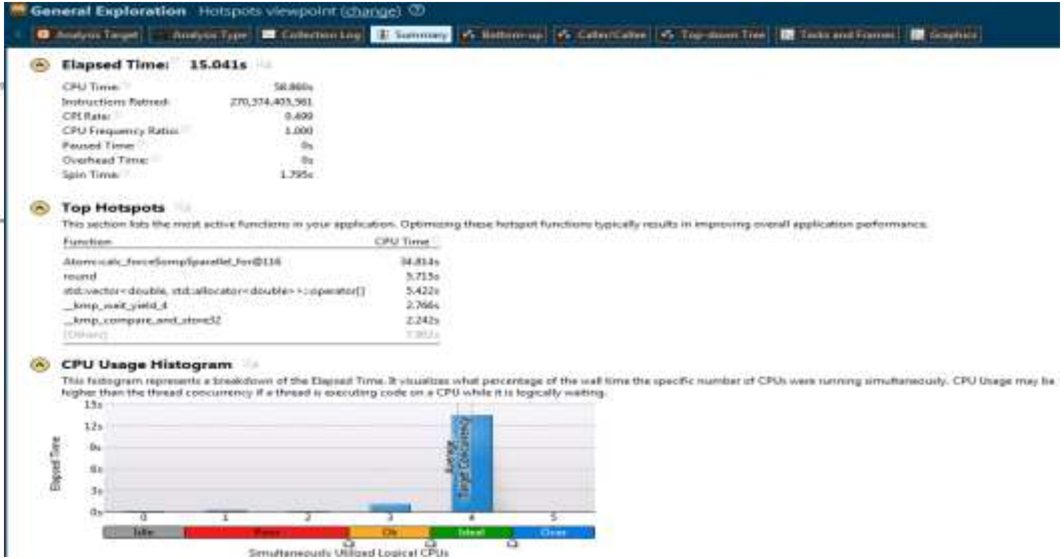
Tuning for Microarchitecture:
- Compiler settings/Vectorization
- Memory/Cache usage
- CPU pitfalls

OS, System — Expertise — SW/uArch

Performance Tuning – Diving Deeper

Perform System and Algorithm tuning first





This presentation uses screenshots from Intel® VTune™ Amplifier XE
The concepts are widely applicable

Algorithm Tuning

A Few Words

- There is no one-size fits all solution to algorithm tuning

- Algorithm changes are often incorporated into the fixes for common issues

- Some considerations:
  - Parallelizable and scalable over fastest serial implementations
  - Compute a little more to save memory and communication
  - Data locality -> vectorization

Compiler Performance Considerations

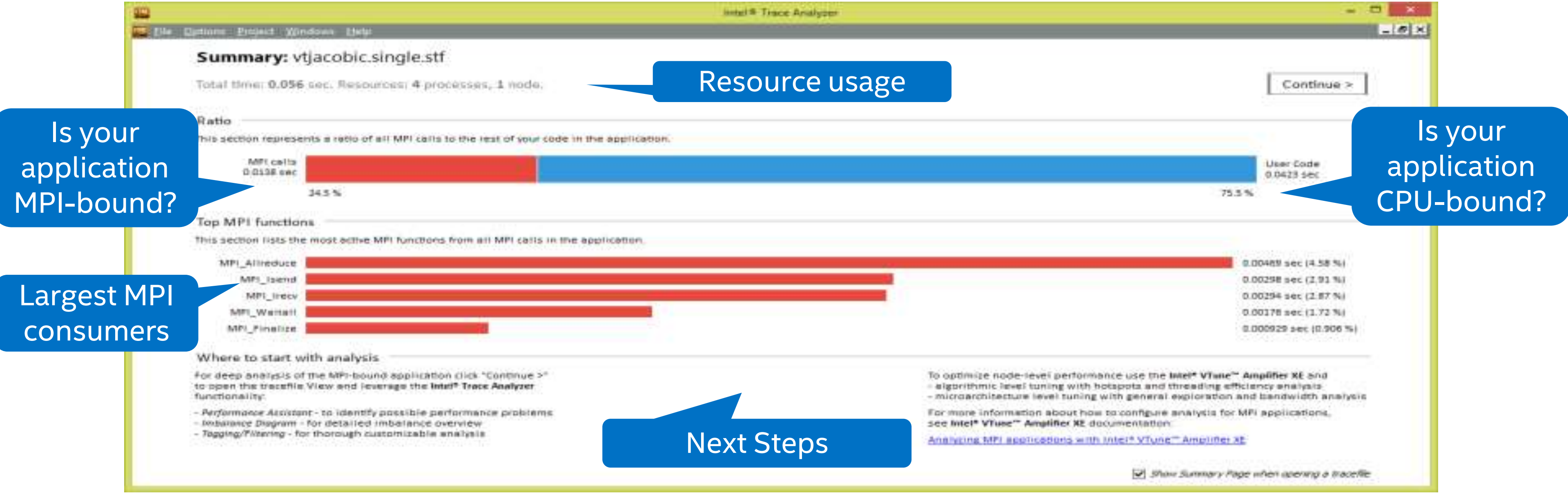
Feature	Flag
Optimization levels	-O0, O1, O2, O3
Vectorization	-xHost, -xavx, etc…
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program **warning: -fast def'n changes over time	-fast (same as: -ipo –O3 -no-prec-div -static -xHost)
Automatic parallelization	-parallel

- Compilers can provide considerable performance gains when used intelligently
- Consider compiling hot libraries and routines with more optimizations
- Always check documentation for accuracy effects
- This could be a day-long talk on its own

This is from the Intel compiler reference, but others are similar

MPI Tuning

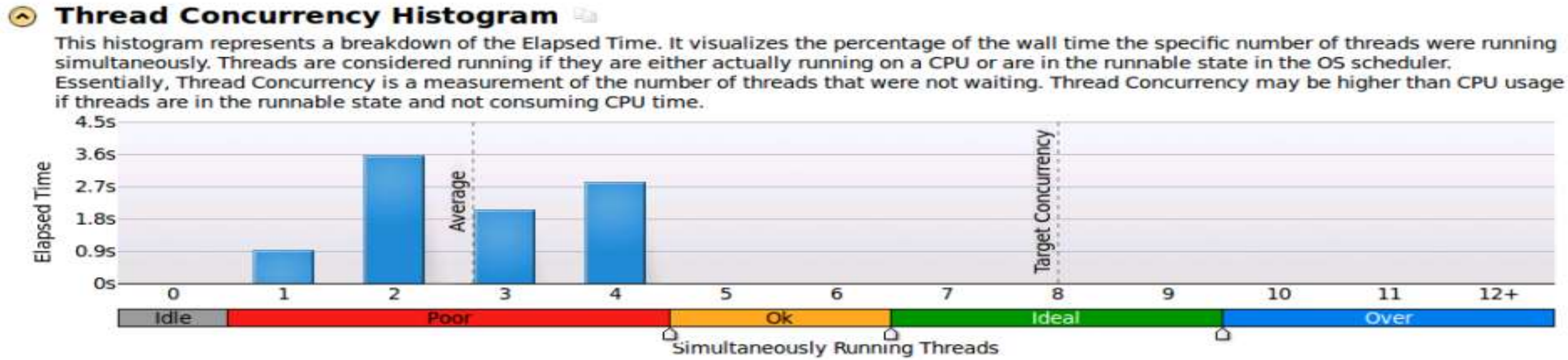
- Find the MPI/OpenMP sweet spot
- Determine how much memory do your ranks/threads share
- Communication and synchronization overhead



Intel® Trace Analyzer and Collector: http://intel.ly/traceanalyzer-collector

Common Scaling Barriers

- Static Thread Scheduling
- Load Imbalance
- Lock Contention



You paid for the nodes, so use them!

Static Thread Scheduling

- Statically determining thread counts does not scale
  - Core counts are trending higher
  - Designs must consider future hardware
  - Commonly found in legacy applications

```
…
NUM_THREADS = 4;
pthread_t threads[NUM_THREADS];
int rc;
long t;
int chunk = limit/NUM_THREADS;
for(t=0;t<NUM_THREADS;t++){
  range *r = new range();
  r->begin = t*chunk;
  r->end = t*chunk+chunk-1;
  rc = pthread_create(&threads[t], NULL, FindPrimes, (void *)r);
}
…
```

Static Thread Scheduling

- Statically determining thread counts does not scale
  - Core counts are trending higher
  - Designs must consider future hardware
  - Commonly found in legacy applications

```
…
NUM_THREADS = 4;
pthread_t threads[NUM_THREADS];
int rc;
long t;
int chunk = limit/NUM_THREADS;
for(t=0;t<NUM_THREADS;t++){
  range *r = new range();
  r->begin = t*chunk;
  r->end = t*chunk+chunk-1;
  rc = pthread_create(&threads[t], NULL, FindPrimes, (void *)r);
}
…
```

Static Thread Scheduling

- Statically determining thread counts does not scale
  - Core counts are trending higher
  - Designs must consider future hardware
  - Commonly found in legacy applications

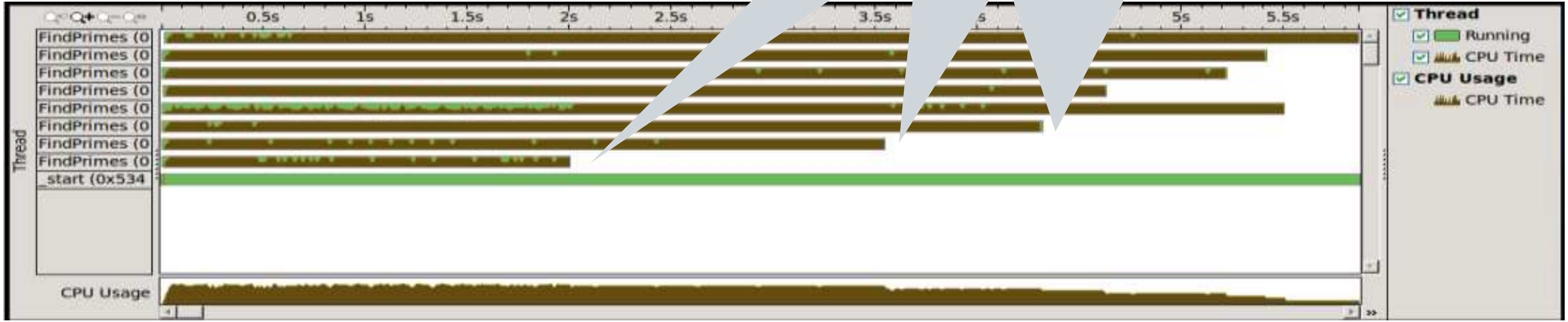
Create Threads Dynamically - `NUM_THREADS = get_num_procs();`

```
…
NUM_THREADS = 4;
pthread_t threads[NUM_THREADS];
int rc;
long t;
int chunk = limit/NUM_THREADS;
for(t=0;t<NUM_THREADS;t++){
  range *r = new range();
  r->begin = t*chunk;
  r->end = t*chunk+chunk-1;
  rc = pthread_create(&threads[t], NULL, FindPrimes, (void *)r);
}
…
```

Load Imbalance

- Dynamically determining thread count helps… but isn't a silver bullet
  - Workload distribution must be intelligent
  - Threads should be kept busy
  - Maximize hardware utilization

Ideally all threads would complete their work at the same time



Load Imbalance

- Dynamically determining thread count helps… but isn’t a silver bullet
  - Workload distribution must be intelligent
  - Threads should be kept busy
  - Maximize hardware utilization

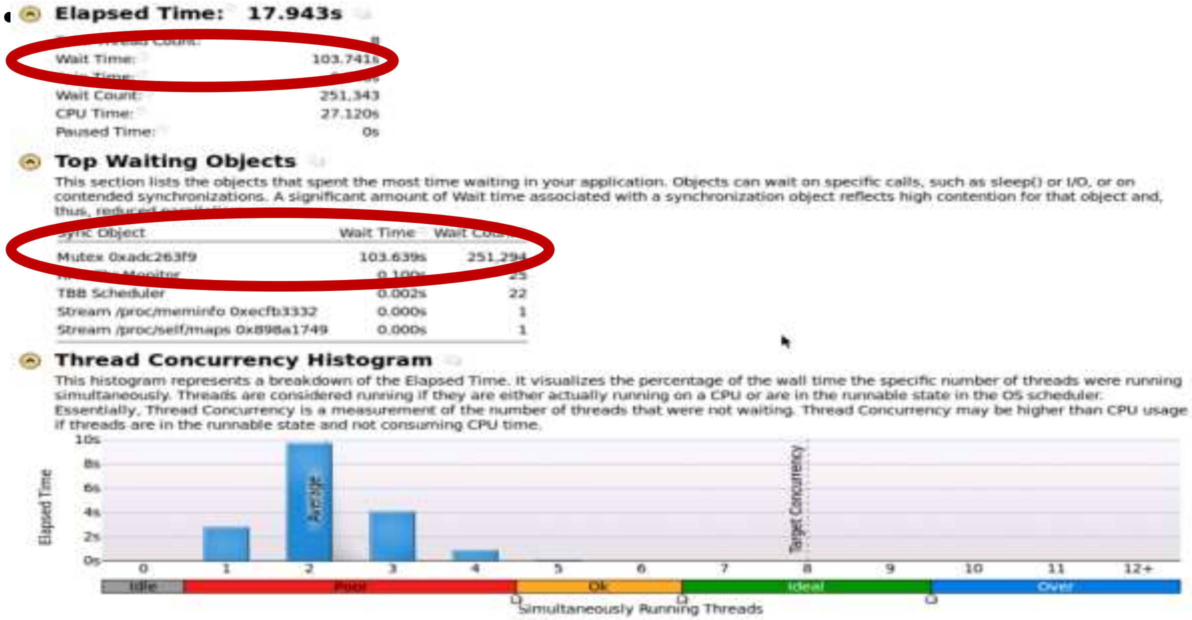
The key to balancing loads is to use a threading model that supports tasking and work stealing

Some examples:

- OpenMP* dynamic scheduling
- Intel Threading® Building Blocks
- Intel® Cilk™ Plus

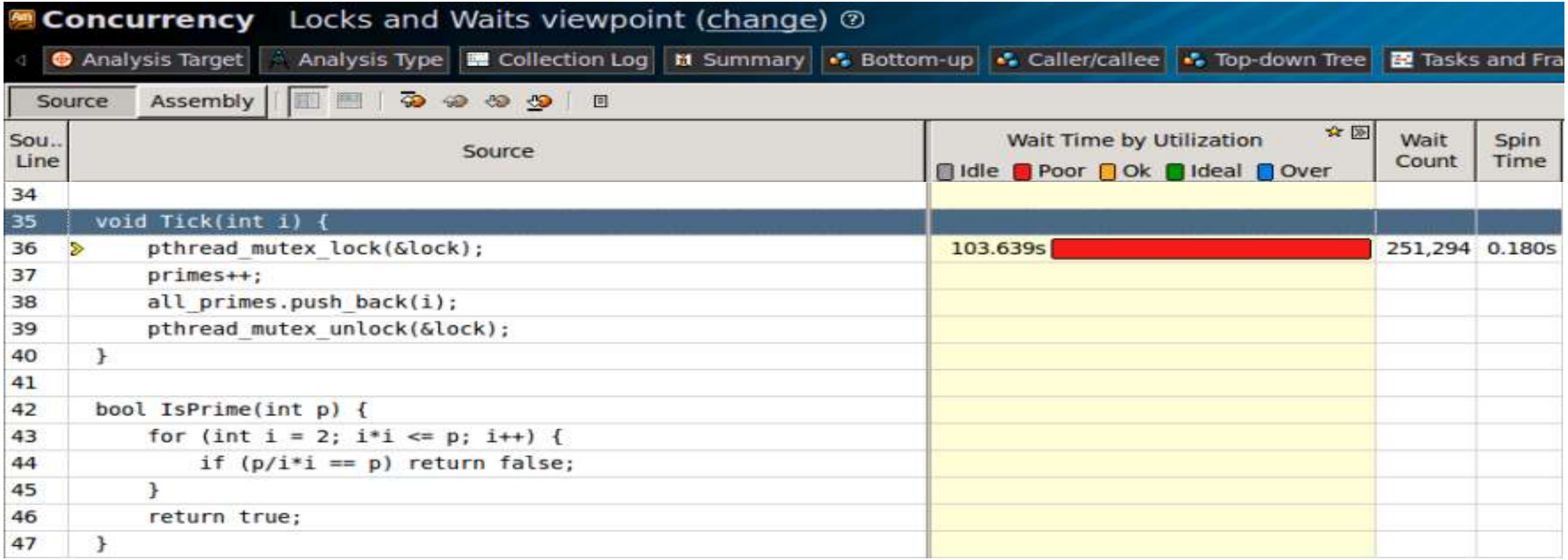
Lock Contention

- A well balanced application can still suffer from shared-resource competition
  - Synchronization is a necessary component
  - Excessive overhead can destroy performance gains



Lock Contention

- A well balanced application can still suffer from shared-resource competition
  - Synchronization is a necessary component
  - Excessive overhead can destroy performance gains
  - Numerous choices for where and how to synchronize



Lock Contention

- A well balanced application can still suffer from shared-resource competition
  - Synchronization is a necessary component
  - Excessive overhead can destroy performance gains
  - Numerous choices for where and how to synchronize

Some solutions to consider:

- Lock granularity
  - Access overhead vs. wait time

- Using lock free or thread safe data structures

```
tbb::atomic<int> primes;
tbb::concurrent_vector<int> all_primes;
```

- Local storage and reductions

Microarchitectural Tuning

Intel uArch specific tuning

After high-level changes look at PMUs for more tuning

- Find tuning guide for your hardware at www.intel.com/vtune-tuning-guides

Every architecture has different events and metrics

We try to keep things as consistent as possible

Start with the **Top-Down Methodology**

  - Integrated with the tuning guides

Introduction to Performance Monitoring Unit (PMU)

Registers on Intel CPUs to count architectural events

- E.g. Instructions, Cache Misses, Branch Mispredict

Events can be counted or sampled

- Sampled events include Instruction Pointer

Raw event counts are difficult to interpret

- Use a tool like VTune or Perf with predefined metrics

Background

Hardware Definitions

Front-end:

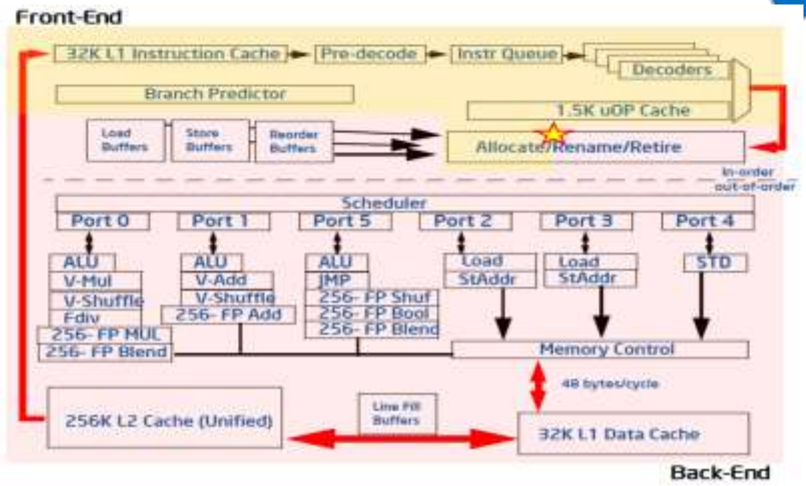
- Fetches the program code
- Decodes them into low-level hardware operations – micro-ops (uops)
- uops are fed to the Back-end in a process called allocation
- Can allocate 4 uops per cycle

Back-end:

- Monitors when a uop's data operands are available
- Executes the uop in an available execution unit
- The completion of a uop's execution is called retirement, and is where results of the uop are committed to the architectural state
- Can retire 4 uops per cycle

Pipeline Slot:

- Represents the hardware resources needed to process one uop



Background

Hardware Definitions

Front-end:

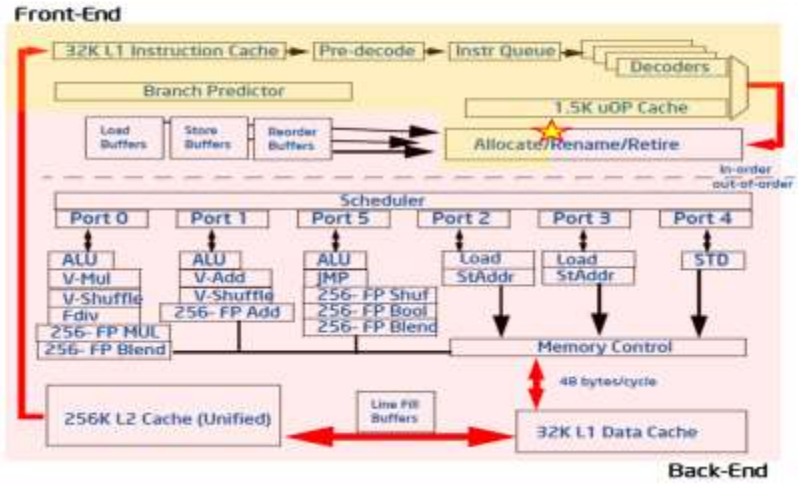
- Fetches the program code
- Decodes them into low-level hardware operations - micro-ops (uops)
- uops are fed to the Back-end in a process called allocation
- Can allocate 4 uops per cycle

Back-end:

- Monitors when a uop's data operands are available
- Executes the uop in an available execution unit
- The completion of a uop's execution is called retirement, and is where results of the uop are committed to the architectural state
- Can retire 4 uops per cycle

Pipeline Slot:

- Represents the hardware resources needed to process one uop



Therefore, modern "Big Core" CPUs have 4 "Pipeline Slots" per cycle

The Top-Down Characterization

Each pipeline slot on each cycle is classified into 1 of 4 categories.

For each slot on each cycle:

The Top-Down Characterization

- Determines the hardware bottleneck in an application
- Sum to 1.0
- Unit is “Percentage of total Pipeline Slots”
- This is the core of the new Top-Down characterization
- Each category is further broken down depending on available events
- Top-Down Characterization White Paper
  - http://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues

Tuning Guide Recommendations

	Expected Range of Pipeline Slots in this Category, for a Hotspot in a *Well-tuned:*		
Category	**Client/ Desktop application**	**Server/ Database/ Distributed application**	**High Performance Computing (HPC) application**
Retiring	20-50%	10-30%	30-70%
Back-End Bound	20-40%	20-60%	20-40%
Front-End Bound	5-10%	10-25%	5-10%
Bad Speculation	5-10%	5-10%	1-5%

Efficiency Method: % Retiring Pipeline Slots

Why: Helps you understand how efficiently your app is using the processors

General Exploration - General Exploration

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

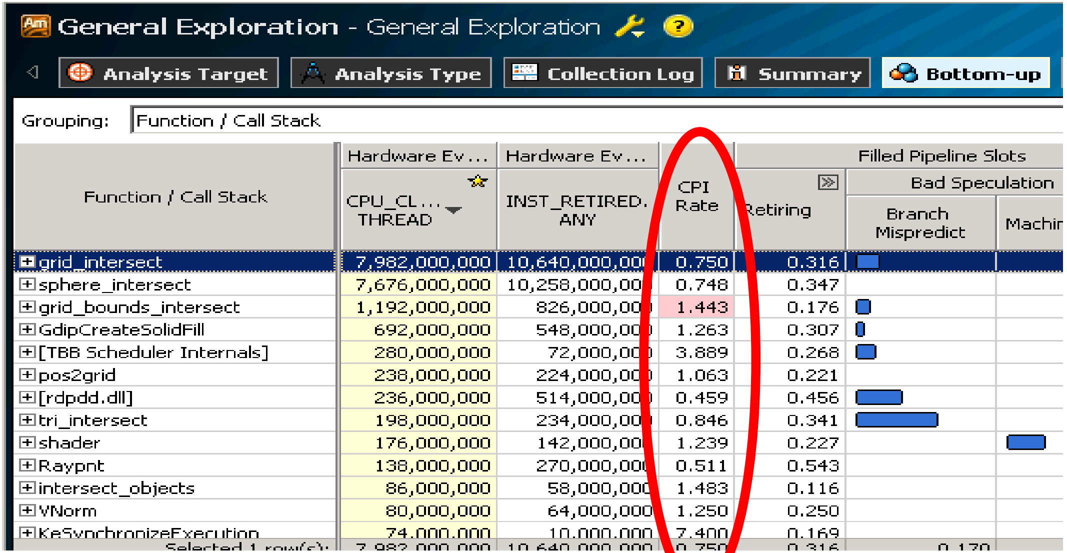
Grouping: Function / Call Stack

Function / Call Stack	Hardware Ev... CPU_CL... THREAD	Hardware Ev... INST_RETIRED.ANY	CPI Rate	Retiring	Filled Pipeline Slots: Bad Speculation: Branch Mispredict	Machir
grid_intersect	7,982,000,000	10,640,000,000	0.750	0.316		
sphere_intersect	7,676,000,000	10,258,000,000	0.748	0.347		
grid_bounds_intersect	1,192,000,000	826,000,000	1.443	0.176		
GdipCreateSolidFill	692,000,000	548,000,000	1.263	0.307		
[TBB Scheduler Internals]	280,000,000	72,000,000	3.889	0.268		
pos2grid	238,000,000	224,000,000	1.063	0.221		
[rdpdd.dll]	236,000,000	514,000,000	0.459	0.456		
tri_intersect	198,000,000	234,000,000	0.846	0.341		
shader	176,000,000	142,000,000	1.239	0.227		
Raypnt	138,000,000	270,000,000	0.511	0.543		
intersect_objects	86,000,000	58,000,000	1.48[illegible]	0.116		
VNorm	80,000,000	64,000,000	1.250	0.250		
KeSynchronizeExecution	74,000,000	10,000,000	7.400	0.169		
Selected 1 row(s):	7,982,000,000	10,640,000,000	0.750	0.316	0.170	

Efficiency Method: Changes in Cycles per Instruction (CPI)

Why: Another measure of efficiency that can be useful when comparing 2 sets of data

- Shows average time it takes one of your workload's instructions to execute



Function / Call Stack	Hardware Ev... CPU_CL... THREAD	Hardware Ev... INST_RETIRED. ANY	CPI Rate	Filled Pipeline Slots Retiring	Bad Speculation Branch Mispredict	Machir
grid_intersect	7,982,000,000	10,640,000,00	0.750	0.316		
sphere_intersect	7,676,000,000	10,258,000,00	0.748	0.347		
grid_bounds_intersect	1,192,000,000	826,000,00	1.443	0.176		
GdipCreateSolidFill	692,000,000	548,000,00	1.263	0.307		
[TBB Scheduler Internals]	280,000,000	72,000,00	3.889	0.268		
pos2grid	238,000,000	224,000,00	1.063	0.221		
[rdpdd.dll]	236,000,000	514,000,00	0.459	0.456		
tri_intersect	198,000,000	234,000,00	0.846	0.341		
shader	176,000,000	142,000,00	1.239	0.227		
Raypnt	138,000,000	270,000,00	0.511	0.543		
intersect_objects	86,000,000	58,000,000	1.483	0.116		
VNorm	80,000,000	64,000,000	1.250	0.250		
KeSynchronizeExecution	74,000,000	10,000,000	7.400	0.169		
Selected 1 row(s):	7,982,000,000	10,640,000,000	0.750	0.316	0.170	

Microarchitectural Tuning - Top-Down

This code is actually pretty good. High retiring percent.

Let's investigate Back-End bound

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

	Hardware Event Count by Har...	Hardware Ev...		Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)	
Function / Call Stack	CPU_CLK_UNHALTED.THREAD	INST_RETIRED.ANY	CPI Rate	Retiring	Bad Speculati...	Back-end Bound	Front-end Bound
Atom::calc_forceompparallel_for@116	79,976,119,964	196,686,295,0 ...	0.407	0.632	0.000	0.355	0.024
round	13,082,019,623	12,624,018,936	1.036	0.344	0.188	0.463	0.006
std::vector<double, std::allocator<double>>::operator[]	12,338,018,507	33,740,050,610	0.366	0.689	0.026	0.251	0.034
__kmp_wait_yield_4	6,448,009,672	3,546,005,319	1.818	0.289	0.003	0.694	0.014
__kmp_compare_and_store32	5,058,007,587	5,440,008,160	0.930	0.298	0.008	0.670	0.024
floor	4,398,006,597	5,096,007,644	0.863	0.425	0.211	0.357	0.006
__kmp_compare_and_store64	2,048,003,072	758,001,137	2.702	0.110	0.018	0.807	0.066

Microarchitectural Tuning - Top-Down

Function / Call Stack	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)				
			Back-end Bound				
				Core Bound			
				Port Utilization			
	Retiring	Bad Speculati...	M. Bo.	Cycles of 0 ...	Cycl...	Cycl...	Cycles of 3+ Ports Ut...
Atom::calc_forceompparallel_for@116	0.632	0.000	0.062	0.082	0.000	0.000	0.411
round	0.344	0.188	0.249	0.175	0.000	0.000	0.565
std::vector<double, std::allocator<double>>::operator[]	0.689	0.026	0.049	0.092	0.000	0.000	0.372
__kmp_wait_yield_4	0.289	0.003	0.451	0.536	0.000	0.000	0.852
__kmp_compare_and_store32	0.298	0.008	0.415	0.527	0.000	0.000	0.738
floor	0.425	0.211	0.152	0.126	0.000	0.000	0.464

Core Bound

This metric shows how core non-memory issues limit the performance when you run out of OOO resources or are saturating certain execution units (for example, using FP-chained long-latency arithmetic operations).

Port Utilization

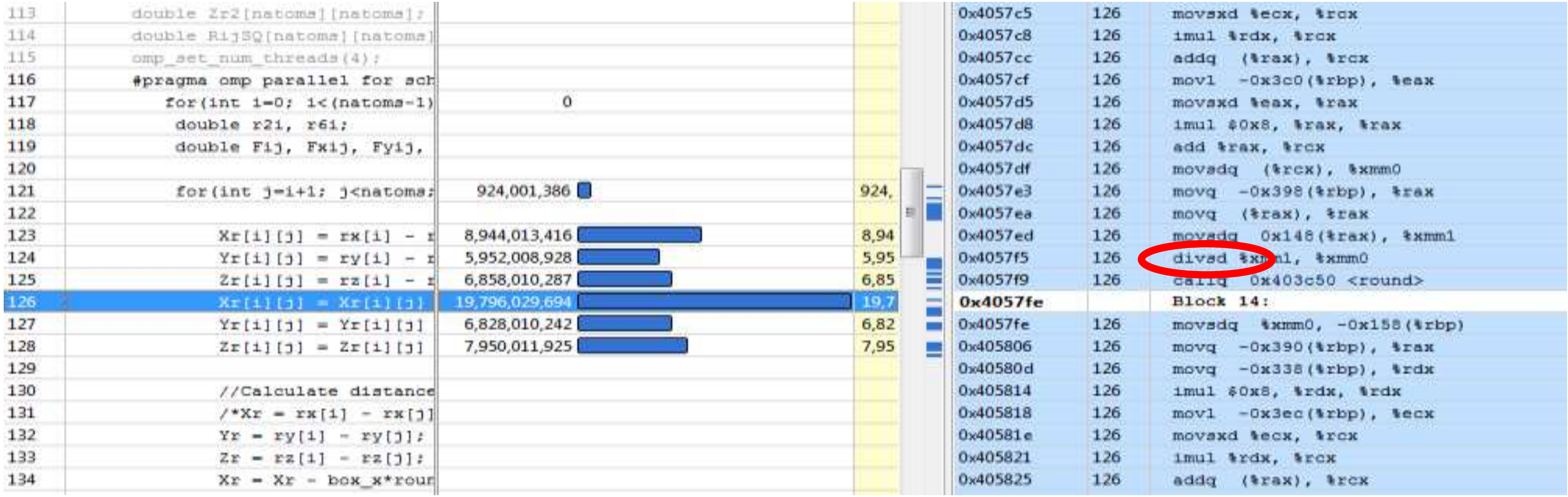
This metric represents a fraction of cycles during which an application was stalled due to Core non-divider-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports.

The number of cycles during which 3 or more ports were utilized.

Threshold: ((((UOPS_EXECUTED.CYCLES_GE_3_UOPS_EXEC) / CPU_CLK_UNHALTED.THREAD) > 0.2) * (CPU_CLK_UNHALTED.THREAD / > 0.05))

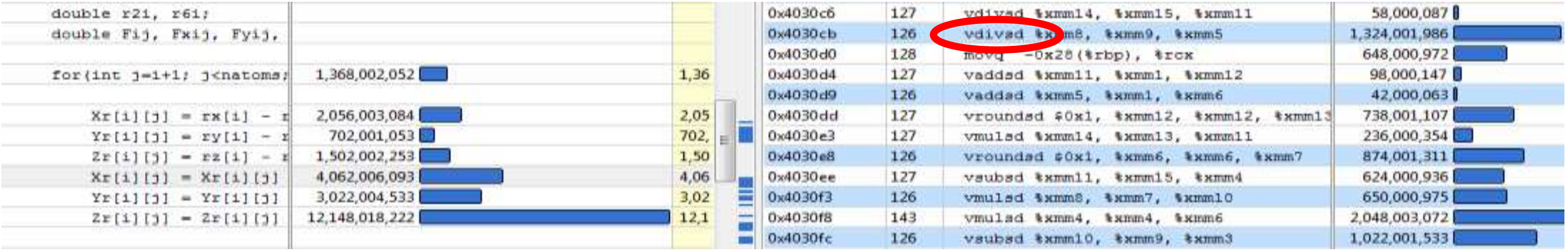
We're basically hammering the compute hardware. Are we vectorizing?

Microarchitectural Tuning - Top-Down



SSE Instructions! Optimize with the compiler e.g. -xhost

Microarchitectural Tuning - Top-Down



AVX2 on Haswell

Before

After

Top-Down with a Memory Bound issue

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	Hardware Event C... CPU_CLK_U... THREAD	Hardware Event... INST_RETIRED.ANY	CPI Rate	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Speculation	Unfilled Pipeline Slots (Stalls): Back-end Bound	Unfilled Pipeline Slots (Stalls): Front-end Bound
multiply1	488,292,732,438	43,100,064,650	11.329	0.022	0.001	0.974	0.003
KeWaitForMultipleObjects	86,000,129	14,000,021	6.143	0.081	0.244	0.430	0.244
KeSetTimer	86,000,129	6,000,009	14.333	0.000	0.000	0.919	0.081

General Exploration General Exploration viewpoint (change)

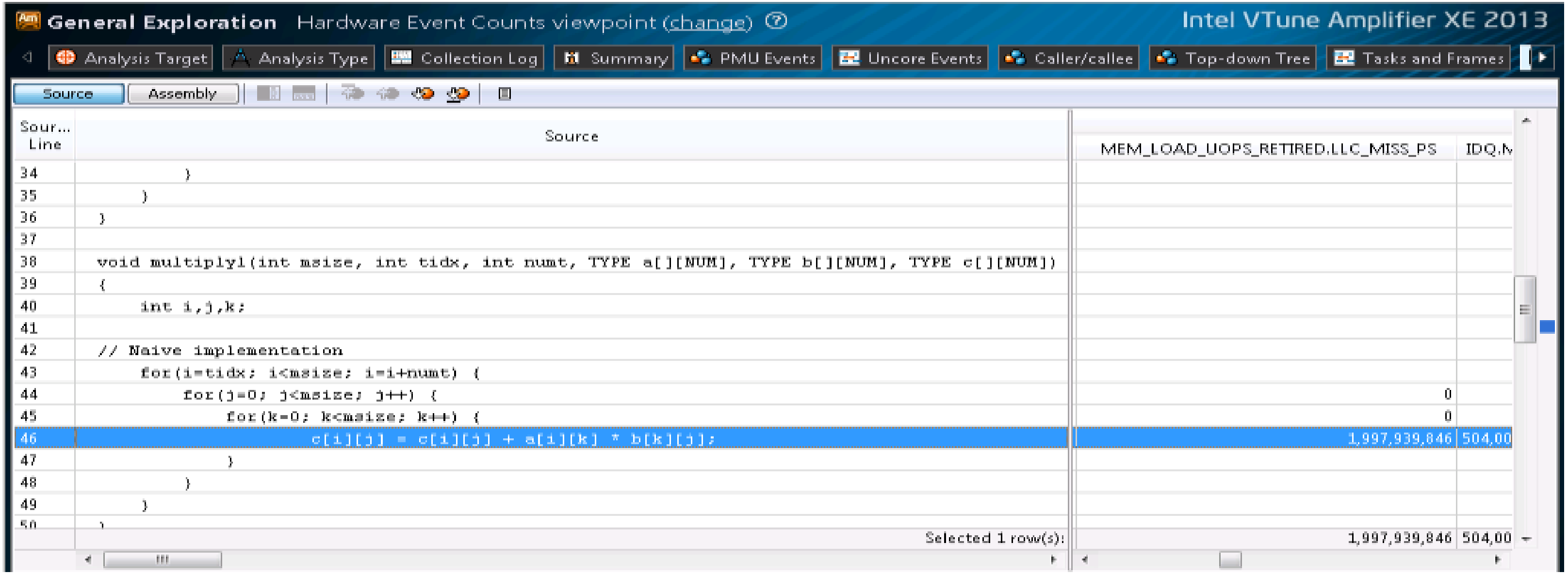
Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Tasks and Frames | mult

Grouping: Function / Call Stack

Function / Call Stack	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)						
			Back-end Bound						
			Memory Bound						
	Retiring	Bad Speculation	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	Core Bound	Front-end Bound
multiply1	0.022	0.001	0.070	0.023	0.064	0.745	0.036	0.022	0.003
KeWaitForMultipleObjects	0.081	0.244	0.000	0.326	0.000	0.000	0.000	0.000	0.244

DRAM Bound Function

Top-Down with a Memory Bound issue



Array accesses are poorly addressed

From Tuning Guide:

- **How:** Memory Bound sub-category, Metrics: *L3 Latency*, *LLC Miss*
- **What Now:**
  - If either metric is highlighted for your hotspot, consider reducing misses:
    - Change your algorithm to reduce data storage
    - Block data accesses to fit into cache
    - Check for sharing issues (See Contested Accesses)
    - Align data for vectorization (and tell your compiler)
    - Use the cacheline replacement analysis outlined in section B.3.4.2 of Intel® 64 and IA-32 Architectures Optimization Reference Manual, section **B.3.4.2**

Top-Down with a Memory Bound issue

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | T

Grouping: Function / Call Stack

Function / Call Stack	Hardware Ev... CPU_CL... THREAD	Hardware Ev... INST_RETIRE... ANY	CPI Rate	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Speculation	Unfilled Pipeline Slots (Stalls): Back-end Bound	Unfilled Pipeline Slots (Stalls): Front-end Bound
multiply2	43,980,065,970	51,604,077,406	0.852	0.353	0.001	0.573	0.073
KeSetTimer	24,000,036	0	0.000	0.000	0.000	1.000	0.000
init_arr	20,000,030	16,000,024	1.250	0.000	0.000	1.000	0.000
KeSynchronizeExecution	18,000,027	0	0.000	0.389	0.000	1.000	0.000
ExReleaseRundownProte	14,000,021	6,000,009	2.333	0.000	0.000	1.000	0.000
Selected 1 row(s):	43,980,065,970	51,604,077,406	0.852	0.353	0.001	0.573	0.073

With a Loop-Interchange (was 97% Back-End bound)

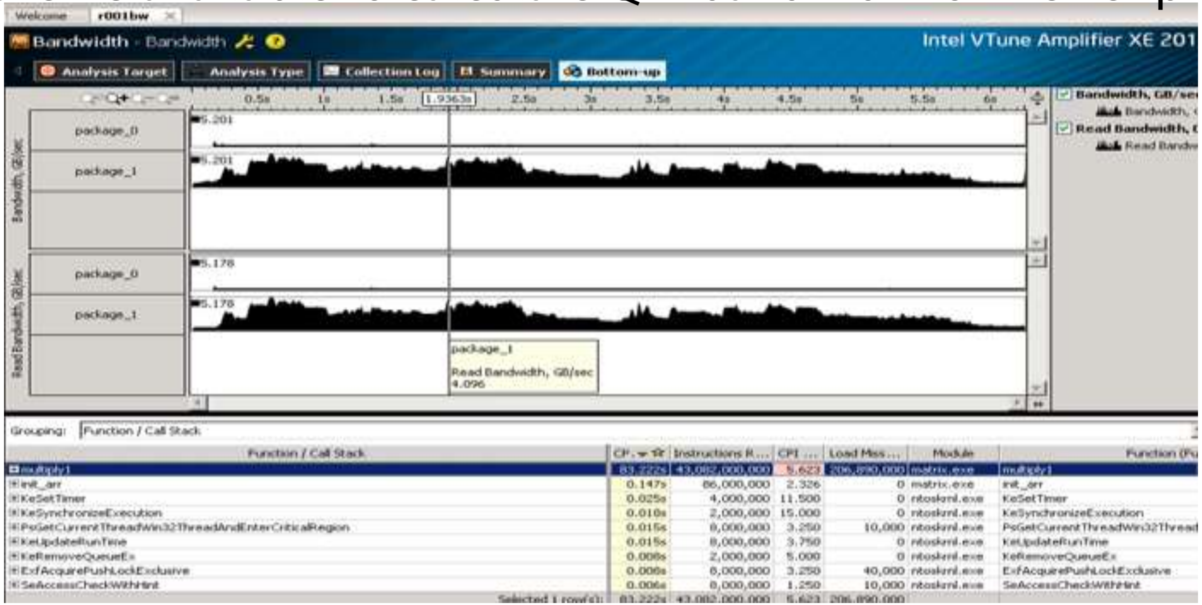
Top-Down for NUMA analysis

Unfilled Pipeline Slots (Stalls)														
Back-end Bound														
Memory Bound													Core Bound	
L1 Bound				Store Bound			L3 Bound			DRAM Bound			DIV Active	Port Utilization
DTLB Ov...	Loads Bl...	Split Loads	4K A...	Fals...	Split...	DTL...	Contest...	Data Shar...	L3 Lat...	Local DRAM	Remote DRA..	Rem...		
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.267
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
0.099	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.411	0.000	0.000	0.000	0.000	0.283
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.444	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.574

- Multi-socket systems with NUMA require special analysis
  - VTune, numastat, numactl
- Remote cache and DRAM accesses can cause stalls
- Now what?
  - Memory allocation vs. access
  - Temporal locality

Memory Bandwidth using PMUs

- Know your max theoretical memory bandwidth
- Locate areas of high LLC misses
- PMU events available to calculate QPI bandwidth on newer processors



Tuning Guides Have Lots of Metrics and Hints

For example:

Back-End Bound

Data Sharing

- **Why:** Sharing clean data (read sharing) among cores (at L2 level) has a penalty at least the first time due to coherency
- **How:** Memory Bound sub-category, Metrics: *Data Sharing*
- **What Now:**
  - If this metric is highlighted for your hotspot, locate the source code line(s) that is generating HITs by viewing the source. Look for the MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS event which will tag to the next instruction after the one that generated the HIT.
  - Then use knowledge of the code to determine if real or false sharing is taking place. Make appropriate fixes:
    - For real sharing, reduce sharing requirements
    - For false sharing, pad variables to cacheline boundaries

Tuning Guides Have Lots of Metrics and Hints

For example:

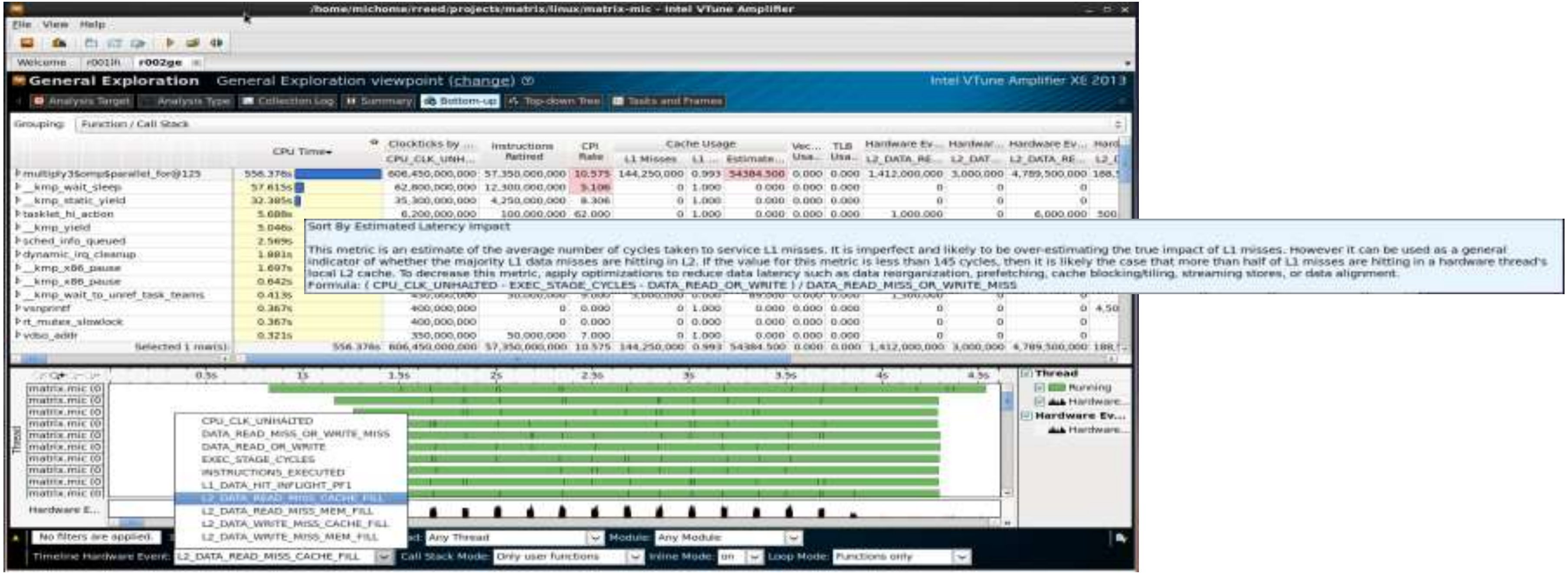
Front-end Latency

Front-End Bound

- **Why:** Front-end latency can lead to the Back-End not having micro-ops to execute (instruction starvation).
- **How:** Front-End Latency sub-category, Metrics: *ITLB Overhead, ICache Misses, Length-Changing Prefixes*
- **What Now:**
  - If any of these metrics are highlighted for your hotspot, try using better code layout and generation techniques:
    - Try using profile-guided optimizations (PGO) with your compiler
    - Use linker ordering techniques (/ORDER on Microsoft's linker or a linker script on gcc)
    - Use switches that reduce code size, such as /O1 or /Os
    - For dynamically generated code, try co-locating hot code, reducing code size, and avoiding indirect calls

Intel Xeon Phi

- Has its own tuning guide and metrics



Intel Xeon Phi

- Efficiency Metric: Compute to Data Access Ratio
  - Measures an application's computational density, and suitability for Intel® Xeon Phi™ coprocessors

Metric	Formula	Investigate if
Vectorization Intensity	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	
L1 Compute to Data Access Ratio	VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE	< Vectorization Intensity
L2 Compute to Data Access Ratio	VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_ WRITE_MISS	< 100x L1 Compute to Data Access Ratio

- Increase computational density through vectorization and reducing data access (see cache issues, also, DATA ALIGNMENT!)

Intel Xeon Phi

- Has its own tuning guide and metrics

- Problem Area: VPU Usage
  - Indicates whether an application is vectorized successfully and efficiently

Metric	Formula	Investigate if
Vectorization Intensity	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	<8 (DP), <16(SP)

- Tuning Suggestions:
  – Use the Compiler vectorization report!
  – For data dependencies preventing vectorization, try using Intel® Cilk™ Plus #pragma SIMD (if safe!)
  – Align data and tell the Compiler!
  – Restructure code if possible: Array notations, AOS->SOA

Performance Optimization Methodology

- Follow performance optimization process
  - Use the Top-down approach to performance optimization
  - Use iterative optimization process
  - Utilize appropriate tools (Intel's or non-Intel)
  - Apply scientific approach when analyzing collected results

- Practice!
  - Performance tuning experience helps achieving better results
  - Right tools help as well

Performance Profiling Tools

Technology wise selection

You have a chose of many:

- From simplest and fastest…

Instrumentation

Sampling

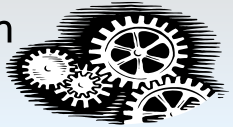
OS embedded:

Task Manager, top, vmstat

- To very complicated and/or slow

Application/platform

Simulators



Project embedded:

Proprietary perf. infrastructure

Always consider overhead vs. level of detail – it's often a tradeoff

Scientific Approach to Analysis

- None of the tools provide exact results
  - Data collection overhead or dropping details
  - Define what results need to be precise
- Low overhead tools provide statistical results
  - Statistical theory is applicable
  - Think of proper sampling frequency (for data bandwidth)
  - Think of proper length of data collection (for process)
  - Think of proper number of experiments and results deviation
- Take into account other processes in a system
  - Anti-virus
  - Daemons and services
  - System processes
- Start early – tune often!

References

- Top-Down Performance Tuning Methodology
  - www.software.intel.com/en-us/articles/de-mystifying-software-performance-optimization
- Top-Down Characterization of Microarchitectural Bottlenecks
  - www.software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues
- Intel® VTune™ Amplifier XE
  - www.intel.ly/vtune-amplifier-xe
- Tuning Guides
  - www.intel.com/vtune-tuning-guides

Look for:

- Confirmation
- Surprises

Do not skip either

Questions?

james.r.reinders@intel.com

Intel Software

James Reinders. Parallel Programming Evangelist. Intel.

James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls Volume 1 (Morgan Kaufmann, Nov. 2014), and High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, Aug. 2015). James is working on a refresh of both the Xeon Phi™ book (original Feb. 2013, revised with KNL information by mid-2016) and a refresh of the TBB book (original June 2007, revised by 2017).

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804